11

Dependability, Fault Detection and Fault Tolerance

11.1 INTRODUCTION

Methodologies for analysing the reliability of complex systems and techniques for making such systems tolerant of faults, thus increasing the reliability, are well established. In hardware the emphasis is on improved components and construction techniques and the use of redundancy to maintain critical systems functions in the event of single failures (in some systems multiple failures). In software the emphasis has been on improved software construction techniques – 'software engineering' – to reduce latent errors; there has also been work on techniques to introduce redundancy into software systems.

The majority of work on fault tolerance has concentrated on what Anderson and Lee (1981) have termed 'anticipated faults', that is faults which the designer can anticipate and hence 'design in' tolerance. A much more difficult and insidious problem is that of faults in the *design* of the system. These are by definition 'unanticipated faults' (and unanticipatable). Design faults can occur in both hardware and software but are more common in software and much of the effort of software engineering has been directed towards reducing design faults, that is unanticipated faults.

Fault tolerance is just one aspect of constructing *dependable* computer systems. Dependability is defined by Laprie as 'that property of a computing system which allows *reliance* to be justifiably placed on the service it delivers' (Laprie, 1989a). He summarises the attributes of dependability as shown in Figure 11.1 An *error* is the consequence of a *fault* in the system and a *failure* is the effect of an error on the service provided by the system. In this chapter we concentrate on the detection of errors and on fault avoidance and fault tolerance. For background work on measures (reliability and availability) see Shooman (1983), Ham (1984), Liebowitz and Carson (1985) and Laprie (1989b).

11.2 USE OF REDUNDANCY

A well-established technique for increasing the reliability of hardware systems is to

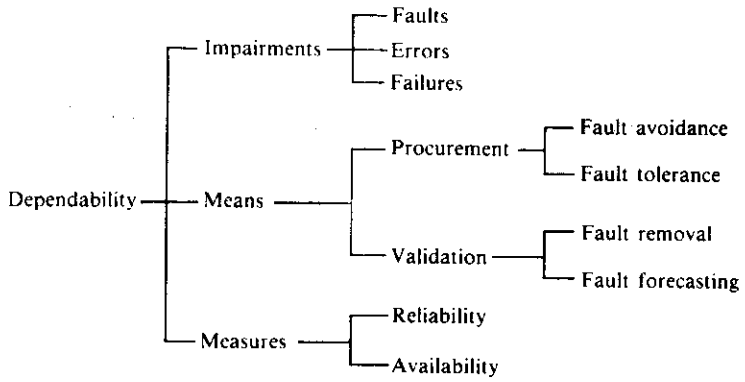

Figure 11.1 Characteristics of dependability.

duplicate or triplicate hardware units. Hardware redundancy can be introduced in two forms: *static* (or masking) redundancy, and *dynamic* redundancy. In static redundancy the duplicate components take over the operation in the event of failure and hence mask the failure of the system from the environment. To detect that a failure has occurred requires some secondary indication of failure. For example, a computer control system with several processors operating in parallel and carrying out (or capable of carrying out) the complete functions of the system with the final action being determined by a majority voting system involves static redundancy. In dynamic redundancy an error detection unit is used to detect an error; once detected the error must be corrected by redundancy elsewhere in the system. An example of dynamic redundancy is a distributed computer system in which each unit carries out a specific subset of the overall functions, but which has some means of detecting when a unit fails and transferring the actions of the failed unit to one or more of the operating units.

The most common example of the static redundancy approach is triple modular redundancy (TMR) in which three identical units are run in parallel and a majority voting system is used to check the outputs. The system is designed to tolerate failure of a single module. It produces output only when two modules agree (2-out-of-3 system).

Although the concepts of multiple channels and majority voting are simple there are some serious practical problems. An obvious problem is that a parallel channel system does not provide any protection against common mode or systematic failures. Another problem is in the decision or voting unit. If the signals entering the unit are two-valued logic signals (Boolean) then a clear unambiguous result is obtained. If any two are true the output will be true; if any two are false then the output will be false. Unambiguous results can normally be obtained also for integer values – they are usually required to be equal. However, if the signals are analog

modulating signals or discrete real numbers then the decision unit is more difficult to implement. Three common techniques are:

Average or mean value: this is simple to implement and can be satisfactory if the signals do not deviate from each other greatly or if a large number of channels are used. For a TMR system a fault which causes one of the signals to go to a maximum or minimum value can introduce a significant error in the output.

Average with automatic self-purging: the average is calculated but if one of the channels deviates by more than a specified amount from the average that channel is disconnected. This technique can generate a large transient when the faulty channel is disconnected.

Median selection: instead of taking the average, the median value of three or more channels is selected. This technique avoids large transients.

In the above it should be noted that only the average with automatic self-purge technique provides detection of failure of a channel.

In any system employing parallel channels there will be divergence, that is the individual channels will not give identical values and also the differences in values may change with time without implying failure of any particular channel. To avoid divergence resulting in disconnection of a functioning channel some form of channel equalisation is necessary (Ham, 1984).

There is a fundamental difference between the use of redundancy in hardware and in software. In hardware it is assumed that each unit functions correctly according to its specification and that failure is due to some physical cause – wear, a faulty component, unusual physical stress – that is not common to both units. In software the major causes of failure are design faults – ranging from misinterpretation of the specification to simple coding errors. Simply replacing a software module which has failed with an exact copy will result in an immediate failure of the replacement copy. The approach required for software redundancy is more complicated requiring the production of independent software modules from the same specification.

The production of independent software modules is not a simple task. It is, for example, not adequate to code independent modules from the same design, or to design blind from the same specification. Experiments have shown that most of the errors in the final system stem from ambiguous or misunderstood specifications. Ambiguity in the specification and misunderstanding can be reduced if the independent groups report back to a co-ordinator.

A danger in using redundancy is that agreement between modules can induce a false sense of confidence in the system, or in parts of the system in which redundancy is used, that can result in the failure to diagnose common mode or systematic errors.

11.3 FAULT TOLERANCE IN MIXED HARDWARE–SOFTWARE SYSTEMS

As indicated above one of the basic techniques for introducing fault tolerance is to add redundancy to the system. However, care is required as adding redundancy adds complexity to a system and complexity can increase the likelihood of faults occurring. Thus great care has to be taken in adding redundancy to a system.

Dealing with the reliability and fault tolerance of mixed hardware/software systems introduces difficulties in determining the actual reliability structure of the system. The software is distributed over several hardware elements, for example a software module driving an input device is distributed over the CPU, memory, interface bus and the input device. Although at the software design stage the input driver is (or should be) clearly defined as a module this structure may not be apparent when the software is run. Anderson and Lee (1981) propose a model in which the hardware is viewed as maintaining an interface on which the software is executed. They term this interface the *interpretative interface* and they represent the model as shown in Figure 11.2, where C is a computing system, S is the memory containing the program (code and data) and H is the rest of the hardware. The

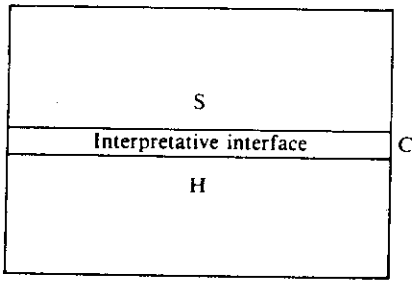

Figure 11.2 Interpretative interface model.

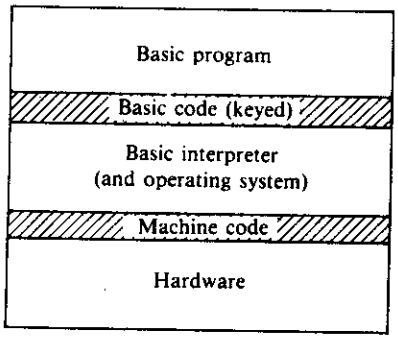

Figure 11.3 Multi-level interpreter system.

interpretative interface represents a language that provides objects and operations to manipulate those objects. At the simplest level this is the machine language of the hardware.

The model is extended in two ways: one is a *multi-level interpreter* system which can be represented as shown in Figure 11.3; the other is the concept of an *extended interpreter* represented as shown in Figure 11.4.

This latter model assumes that the system is required to support an interpreter L_1; however, what is available is some interpreter (language) L_0. If L_0 contains some of the facilities required by L_1 then it is not necessary to provide a complete interpreter; an interpreter which uses some of L_0 but extends it to provide extra facilities can be used. The program can then be considered as being in two parts: part E which is written in L_0 and part P which is written in L_1. The interpreter extension model clearly can be used to represent operating systems. It can be extended to represent *multi-level extended interpreters.*

11.3.1 Mechanisms and Measures

In using this modelling approach two further concepts are helpful. They are used to distinguish between facilities provided by the interpreter and those which are programmed in the interpreted system:

- *mechanism*: provides a specific facility at the interpreter interface and is implemented as part of the interpreter;
- *measure*: performs some specific task and is implemented by means of a set of instructions in the system that is interpreted.

Consider the input of data into a program written in BASIC and suppose that the program has to accept integer values. If it is written using the standard input with a BASIC integer variable given as part of the input statement, then if the user inputs alphabetic characters the error will be detected by a *mechanism* contained within the BASIC interpreter which will force an entry to the error handling system. The programmer will thus have relied on an interpreter mechanism. Alternatively if the input is read into a string variable and string to numeric conversion is performed by a program, then input checking is performed using a *measure*, not a *mechanism*. Mechanisms are likely to be more general purpose and widely applicable than measures and Anderson and Lee place emphasis on generating fault tolerance through the use of mechanisms.

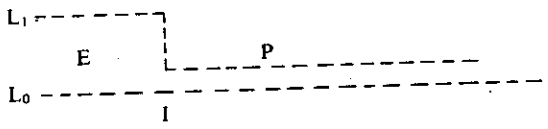

Figure 11.4 Extended interpreter system.

11.3.2 Exceptions

A second basic idea is to separate the normal behaviour of a system from abnormal behaviour. It is assumed that a fault condition will generate an error which will be an exception. A way of reducing complexity is to separate the response to an event into a normal response and an exception response. System designers and implementors should be able to deal with the two responses separately.

For example, a programmer does not normally have to check for 'divide by zero'; the computer hardware is designed to detect this fault and generate an exception response. The normal exception response in this case is a software trap which transfers control to an error handling routine. In many systems the error handler has a fixed, predetermined response – to halt execution of the program. For real-time systems the fixed response is often not wanted and if it cannot be disabled the designer is forced to insert additional coding to trap errors prior to them initiating the built-in mechanism. What is required is a system which allows the designer explicitly to invoke an exception mechanism in response to some predetermined event. This is generally referred to as *raising* an exception. Facilities for enabling and disabling particular handlers are also needed.

A number of high-level languages provide exception handling mechanisms; some of the mechanisms provided by BASIC interpreters designed for real-time applications are both simple and effective. They permit the designer to execute specific instructions in the event of predicted exceptions. A more complex mechanism is provided in the Ada language: when an exception is raised the system backtracks up the calling structure tree until it finds a handler for the specific exception; if no handler has been provided by the implementor then the exception is passed to the built-in run-time handler.

In distributed systems exception handling mechanisms can become complex if handlers are distributed across different processors. The effect of such a distribution is to close-couple the system and hence detract from the improved robustness that can be obtained from a loosely coupled distributed system.

11.4 FAULT DETECTION MEASURES

11.4.1 Replication Checks

The use of replication checks involves duplicating or replicating the activity of the system that is to be checked. A typical example is the use of dual processors each of which runs the system software. The actions of the two systems are compared at specific times or in response to specific events. If it is assumed that the system design is correct and that failures will be independent then no single fault will go undetected. With replication some additional operation is necessary to determine which of the two units is faulty.

The procedure can be extended to include triple redundancy (TMR) and multiple redundancy (*N*-modular redundancy, NMR). Higher orders of redundancy do not increase the error detection capability but they enable the faulty unit to be identified and the system to continue to run.

Replication is expensive and can increase the complexity of the system. Both can be reduced by limiting replication to selected critical functions. Replication using identical software units cannot detect common mode faults due, say, to errors in the design of the system. To check for design faults the duplicate system would have to be a completely independent design. Adopting such a technique is costly. An alternative is to use a model of the system, for example a Petri-net-based model, to check the high-level control flow of the operational system. Replication can be used in a simple way to detect transient errors by repeated operation of the same system. This technique is frequently used in communication systems and in software for accessing disk drives where several attempts will be made to read or write to a disk before reporting an error and halting the action.

A major problem with replication checks is in deciding on the nature of the comparison. For Boolean and integer values comparison is simple – exact matching is required. For real values some error range must be specified: what should it be? Are cosmetic variations, for example capitalisation and spacing, permitted in strings? For distributed systems where, for example, the replication checks may be performed on different processors, should the data be transmitted in binary form or as ASCII characters? What about complex data structures: do they have to match as a whole or element by element? This is important in triple replication as a structure containing two elements $\{x,y\}$ would generate an error using matching as a whole if the three results returned were $\{a,b\}$, $\{a,d\}$ and $\{e,b\}$ but would return a consensus result $\{a,b\}$ on an element-by-element comparison.

11.4.2 Expected Value Methods

The majority of methods used to detect software errors make use of some form of expected value. The correct progression of a program through a sequence of modules can be checked by using 'baton' passing. Each module passes to its successor module a unique numerical value. If the value received by the successor module does not match the expected value then the sequence of modules is incorrect.

Another technique, sometimes referred to as an 'assertion test', is to build into the program logical tests. Range checking and input/output checks offered in many modern compilers automatically insert coding for assertion testing. Assertion tests can also be incorporated into the application program explicitly by the designer. For example, if on the basis of the software specification, an array or table should at some particular point contain only positive values, then a test to check that this is the case can by inserted in the program code.

Because of the overhead which this type of checking incurs it is often restricted to the testing period. Many compilers provide switches to allow the range checking

and other forms of checking to be selected at compile time. Similarly other forms of checking code can be put into conditional compilation segments to allow it to be omitted in the delivered software system. Careful use of assertion testing on entry to and exit from modules can help to stop a fault generated in one module giving rise to faults in other modules.

Adoption of a 'good' module division at the design stage enables 'defensive programming' to be used to check data passing between modules. It is strongly recommended that data passing between modules be carefully checked. If the design has been well done, the properties of data at module boundaries should have been carefully described and documented and hence checks should be capable of giving a clear indication of any faults. For distributed systems expected value techniques should be used as a method for checking information passing between processors. By using such checks the propagation of errors arising from faults can be prevented. For real-time control and data collection applications use of knowledge of the environment enables expected value methods to be extended to include limit and trend checking as well as techniques based on prediction and analysis of the incoming data from the plant.

11.4.3 Watch-dog Timers

Timing checks, that is checks that some function has been carried out within a specified time, can be used to detect errors in a system. It should be noted that timing checks do not indicate that a system is functioning correctly; they can only indicate that it has failed in some way.

Watch-dog timers are commonly used to monitor the detailed behaviour of a real-time system. A watch-dog task `'watchdog_timer'` is run at fixed time intervals as a high-priority task. It decrements counters (timers) held in a pool and checks that the counters have not reached zero. If a counter has reached zero the `'watchdog_timer'` task signals to the executive that a time out has occurred. Tasks being monitored by the watch-dog timers periodically reset the counters; thus a time out occurs when they fail to reset the counter within a given time interval. Watch-dog timers can also be used to monitor peripherals. By monitoring critical tasks they can also show up a temporary overload on the system, or clearly show that a system has insufficient processing capacity.

The watch-dog timer can also be applied to software communication channels. Problems of synchronisation can often be simplified if a task can restrict its commitment to waiting for information for a predetermined time. In many real-time applications there are circumstances where it is better to continue with out-of-date information (for example, the previous value) or to estimate a data value rather than simply to wait indefinitely. For example, in a feedback control system it is normally better to continue, in the event of a failure of a single instrument, with an estimated value than to halt the controller action. The failure must of course be reported and the instrument repaired or replaced within a reasonable time.

11.4.4 Reversal Checks

For systems in which there is a one-to-one relationship between input and output a useful error check is to compute what the input should be for the actual output and compare the computed input value with the actual input value. For example, disk drivers often read back the data segment which has just been written and compare the data read back with that which was sent. Mathematical computations often lend themselves to reversal checks; an obvious example is to check the computation of a square root by squaring the answer. However, care is needed in such checks to take into account the finite accuracy of the computation.

A variation on this technique can be used in systems in which there is a fixed relationship between the output and input. For example, for matrix inversion the product of the input matrix and the output matrix should be the unit matrix.

11.4.5 Parity and Error Coding Checks

Parity and error coding checks are well-known techniques for detecting and correcting specific types of memory and data transmission errors. The most widely known is the single-bit parity check applied to memory storage and to asynchronous data transmission. This will detect the loss of an odd number of bits in a storage or transmission unit. A simple parity is an effective and efficient solution to detecting errors when they occur as single-bit errors in a unit; it is not an effective method if typical failures result in multiple-bit errors.

More complicated codes such as the Hamming, cyclic redundancy, and M-out-of-N codes are used to detect multiple-bit errors and some of these codes allow reconstruction, that is correction of the error. The use of these codes involves adding a greater amount of redundancy than the use of a simple parity code.

The above codes are generally used to detect errors due to hardware component failure. Codes can also be applied to data to detect software errors. For example, the use of a checksum computed on a block of data and held with it can be used to detect both hardware and software errors. A combination of parity and checksum added to a data unit and a block of data respectively provides an efficient method of error detection for large and complex sets of data.

11.4.6 Structural Checks

Checksum codes are typically applied to data structures and provide a check on the consistency and integrity of the data contained in the structure. Errors can also occur in the data structure itself, for example pointers in linked lists can be corrupted. Thus checks on the structural integrity are also required.

A very simple check in a linked list is to maintain in the list header a pointer to the last item in the list as well as one to the first item. If the pointer to the last item does not correspond to the last item identified by following through the chain of pointers from the first item then an error has occurred. A simple check of this form, however, provides little information on the error and the possible damage. Also there is little information to help with recovery. A frequently used alternative scheme is the doubly linked list in which each element contains both a forward and a backward pointer.

11.4.7 Diagnostic Checks

Diagnostic checks are used to test the behaviour of components used to construct the systems, not the system itself. The checks use a set of inputs for which the correct outputs are known. They are usually programs used to test for faults in the hardware of the system and typically are expensive in terms of the time and resources used. As a consequence they are rarely used as primary error detection measures but are restricted to attempts to locate more precisely faults detected by other means. A difficulty widely experienced is that it is not easy to design diagnostic checks that impose on a component conditions as stringent as those imposed by the actual system.

11.5 FAULT DETECTION MECHANISMS

Some mechanisms for fault detection based on interface exceptions have already been mentioned. These include illegal instruction, arithmetic overflow and underflow, protection violation and non-existent memory. Few systems offer mechanisms beyond these. Although, for example, a compiler for a strongly typed language will detect and flag as errors attempts to perform arithmetic operations on variables declared of type character, the underlying hardware will not distinguish between words holding integers, characters, reals, etc. Hence a run-time fault that results in an attempt to add a word containing a character to a word containing a real would not result in an exception being signalled. One mechanism to overcome this problem, the use of tagged storage, has been offered on a few computer systems. With tagged storage a few extra bits in each word are allocated to be used to identify the type of object being held in the word.

Other useful mechanisms for fault detection are ones which detect attempts to exceed array bounds or to exceed ranges specified for integers and these are provided in a few computers.

11.6 DAMAGE CONTAINMENT AND ASSESSMENT

Damage results from faults or errors propagating throughout the system. A major problem in assessing the extent of the damage arises through the presence of parallel operations. This is true for both standard sequential programs running on a single processor and for multi-tasking and multi-processor distributed systems. The larger the number of tasks and processors the greater the problem.

One approach to the problem is to use techniques similar to those used for preventing clashes between competing concurrent processes. Sections of code are treated as critical sections, or certain objects are permitted to belong only to one task at any particular time. Access control mechanisms – monitors, guards, locks – are used to protect critical sections or objects.

Obvious containment measures are to limit access to a file while it is being updated by a task. If checks on data consistency and structural checks are carried out prior to releasing the file then the effects of a faulty update can be prevented from spreading.

An important technique for assessing damage is to mark bad data in some way. This technique can also be used for containment if incoming data to a task is checked to see if it is marked as bad. In distributed systems it is important to test thoroughly all incoming data messages since the detection of errors or bad data at the input can prevent damage spreading between processors.

11.7 PROVISION OF FAULT TOLERANCE

11.7.1 Redundancy

A method for introducing redundant modules into a system is the use of *recovery blocks*. Associated with a check point is a recovery block which contains one or more alternative code modules. If an error is detected the primary alternative is used. At the end of the recovery block there is an acceptance test: if, as a result of running the alternative code module, the acceptance test is passed the recovery block is exited. If the acceptance test fails then the next alternative module is tried. If all code modules are exhausted before an acceptable result is achieved then an error is reported.

The general structure is:

```
establish recovery point
primary module
acceptance test
alternate module
acceptance test
```

and this is normally expressed using the syntax:

```
ensure <acceptance test>
by          <primary module>
else by<alternate module 1>
else by<alternate module 2>
.
.
.
else by<alternate module n>
else error
```

A characteristic of real-time systems that can be used to simplify the problem of fault recovery is that tasks are repeated at frequent intervals with new data sets supplied at the system boundary. For example, for a feedback control loop a single set of bad data values can be ignored if it can be assumed that the next set of readings will be correct. If the control loop is stable and well designed the effect will be minor. The effect of a single bad set can be reduced further by replacing the bad set with a predicted value based on some form of extrapolation. The use of predicted values can allow for a series of bad values until some other corrective action can be taken, for example switching to another instrument.

A simple recovery block could be:

```
ensure <data good>
by      <normal value module>
else by<predicted value module>
else error
```

A more complex error recovery block would need to take into account in the acceptance test the time for which the data was bad, that is for how long it is acceptable to use predicted data. And also in practice care would be required to avoid generating a spurious error if the next good data value had diverged from the predicted value.

11.7.2 Deadline Mechanisms

The provision of fault tolerance for real-time computer systems has to take into account that the time to perform some operation enters into the specification of the system and failure to complete in some specified time constitutes an error. Therefore there may not be time to carry out some of the recovery procedures outlined above unless special techniques are used. One such technique is the *deadline mechanism*.

This is an attempt to deal with the requirement that service must be completed within a specified time. The problem is illustrated in the following code segment for

a fault-tolerant navigation system:

```
every 1 second
within 10 milliseconds
calculate by
     (* primary module *)
     read sensors
     calculate new position
else by
     (* alternate module *)
     approximate new position from
     old position
```

The `every` statement is used to specify the repeat time of the particular task. The `within` statement specifies the maximum amount of time that can be permitted to elapse between starting the task and getting the results back from it. Two modules are specified, a primary module which reads the sensors and calculates the new position and a single alternate module which estimates the position from the previous position value. It is assumed that the alternate module is error free and requires less time to run than the primary module.

In order to meet the overall time deadline the system implementor has to determine accurately the execution time for the alternate module in order to determine how long can be allowed for the primary module to produce a good result. For the 10 ms deadline if the alternate module is estimated to take 3 ms then the primary module must return a result within 7 ms.

11.7.3 Bad Data Marking

The use of modular design techniques facilitates the use of defensive programming to detect bad or faulty data. The major problem arises not in detecting bad data but in deciding what action to take. An interesting solution to this problem is that adopted by the designers of the CUTLASS system. Associated with each data variable is a tag bit which indicates whether or not the data is bad. When a test on the data is carried out the tag bit is set or reset according to the result of the test. Bad data is allowed to propagate through the system but carries with it an indication that it is bad. The language support system contains known rules for evaluating expressions containing bad data. For example, if two logic signals are combined using the OR function then both have to be bad for the result to be tagged as bad, whereas for the AND combination if either signal is bad then the result is tagged as bad. With this approach the program module which determines that a data value is bad does not need to know which modules it should inform since the information is automatically conveyed with the data.

CUTLASS was designed to support distributed control and the use of tagged data avoids some of the problems of synchronising recovery in distributed systems.

The tagging of data allows easy implementation of recovery blocks and gradual degradation of the system as a particular task can be programmed to select alternate function modules according to the status of the input data set.

For example, if the control signal calculated for a particular actuator is tagged as bad the output module can be programmed to leave the actuator set at its last value, move the actuator to a predicted value, or move the actuator to a predicted value unless the bad data time exceeds a preset amount at which time the actuator position is frozen. In the recovery block notation this can be expressed as:

```
ensure          <data good, timeout=false>
by
   ensure       <data good>
      by        <primary module - output data>
      else by   <predict output>
   else error
else by         <do not move actuator>
else error
```

The module in the inner block used to predict the output needs to have an acceptance check in it to avoid large movements of the value if the bad data indication comes at a time when a large-value movement is occurring. Such an acceptance check could force an error which would cause the outer block to institute the `<do not move actuator>` module.

11.7.4 Recovery Measures – Check Points

A standard technique which has been used for a long time in data processing systems is to insert check points in the program. At a check point a copy of data which would be required to restart the program at that point is written to backing store or to some protected area of memory. If a fault is detected at a test point or the next check point in the program then the program is 'rolled back' to the previous check point and restarted.

The strategy can be readily adapted for use in real-time systems. In a control system, for example, back-up copies of values such as set points, controller parameters and possibly a 'history' of selected plant variables are held either on backing store or in battery backed-up memory. In the event of failure they can be reloaded.

Recovery mechanisms involving the storage of data values can provide protection against hardware faults or software faults which do not recur. However, in the majority of cases if a software module fails once it will continue to fail. The only way in which the system can continue is if a replacement module can be activated. This technique implies redundancy.

11.8 SUMMARY

We have described some of the problems of designing and producing reliable software and how these problems are different from those associated with hardware reliability. The importance of being able to produce reliable and fault-tolerant software will continue to increase as the use of devices incorporating computers grows. Improved production methods will increase software reliability; however, it will remain difficult to make software-based systems tolerant of unanticipated errors. Safe failure of systems incorporating software will continue to depend in the last resort on hardware safety provisions.

For many real-time engineering applications loosely coupled distributed computer systems can be used. The advantage of such systems is that they can be made robust through the use of redundant processors and communication systems. The loose coupling permits the containment of faults to specific parts of the system thus reducing the extent of the performance failure and easing the problem of reinstatement. Most applications of this type are well defined and hence it is possible to specify closely actions to be taken in the event of expected failure modes and most systems can continue to operate in a degraded mode. For example, it is normal for tasks to be allocated at construction time to a specific node (processor) in the system and the action to be taken in the event of failure of that node is normally predetermined.

As with any distributed system there is the problem of system consistency and steps must be taken to ensure that at the end of the recovery procedure the system is returned to a consistent state. For many real-time engineering applications much of the data decreases in value as it ages and hence much of the system will automatically return to a consistent state as new plant data is obtained. The problem areas concern environmental data that is input in incremental rather than absolute form; in such cases consistency can only be regained by restarting the device or system from some known absolute datum.

Bibliography

ABBOTT, C., 'Intervention schedule for real-time programming', *IEEE Trans. Software Engineering*, SE-10(3): 268–74 (1984).

AHSON, S.I., 'A microprocessor-based multi-loop process controller', *IEEE Trans. Industrial Engineering*, IE-30(1): 34–9 (1983).

ALEXANDER, H., JONES, V., *Software Design and Prototyping Using 'Me too'*, Prentice Hall, Englewood Cliffs, NJ (1990).

ALFORD, M.W., 'A requirements engineering methodology', *IEEE Trans. Software Engineering*, SE-3: 180–93 (1977).

ALLWORTH, S.T., ZOBEL, R.N., *Introduction to Real-time Software Design* (2nd edition), Macmillan, London (1987).

ANDERSON, T. (editor), *Resilient Computing Systems*, Wiley, New York (1985).

ANDERSON, T., LEE, P.A., *Fault Tolerance, Principles and Practice*, Prentice Hall, Englewood Cliffs, NJ (1981).

ANDERSON, T., RANDELL, B. (editors), *Computer Systems Reliability*, Cambridge University Press, Cambridge (1979).

ANDREWS, G.R., 'Synchronising resources', *ACM Trans. Programming Languages and Systems*, 3(4): 405–31 (1981).

ANDREWS, G.R., 'The distributed programming language SR – mechanisms, design and implementation', *Software Practice and Experience*, 12(8): 719–54 (1982).

ANDREWS, M., *Programming Microprocessor Interfaces for Control and Instrumentation*, Prentice Hall, Englewood Cliffs, NJ (1982).

ANON, 'Computing control – a commercial reality', *Control Engineering*, 9(5): 40 (1959).

ARZEN, K.E., 'An architecture for expert system based feedback control', *Artificial Intelligence in Real-Time Control, IFAC*, pp. 15–20 (1988).

ASTROM, K.J., WITTENMARK, B., *Computer Controlled Systems: Theory and design*, Prentice Hall, Englewood Cliffs, NJ (1984).

ASTROM, K.J., ANTON, J.J., ARZEN, K.E., 'Expert control', *Automatica*, 22(3): 277–86 (1986).

AURICOSTE, J.G., 'Applications of digital computers to process control', in Coales, J. (editor) *Automation in the Chemical, Oil and Metallurgical Industries*, Butterworth, London (1963).

AUSLANDER, D.M., SAGUES, P., *Microprocessors for Measurement and Control*, Osborne/McGraw-Hill, New York (1981).

AUSLANDER, D.M., TAKAHASHI, Y., TOMIZUKA, M., 'The next generation of single loop